EFFICIENT STATE MANAGEMENT TO SPEED UP SIMULTANEOUS SIMULATION OF ALTERNATE SYSTEM CONFIGURATIONS

Shravan Gaonkar, Tod Courtney and William H. Sanders

Coordinated Science Laboratory and Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign Urbana, IL 61801, U.S.A.
{gaonkar, tod, whs}@crhc.uiuc.edu

ABSTRACT

Simulation is a useful tool for engineering systems, and it is used in numerous and diverse fields. Simulation of large systems can result in better designs, and make the design process more efficient. One important use of simulation lies in comparing different simulation models that represent competing or alternative designs before the actual deployment and implementation. Simultaneous Simulation of Alternative System Configurations (SSASC) provides a methodology that exploits the structural similarity among the alternative configurations and results in an efficient simulation algorithm that evaluates alternative configurations of a system simultaneously. In this paper, we designed and implemented an efficient data structure for simulation state management to speed up SSASC. This approach is orthogonal to parallel or distributed simulation. The resulting improved algorithm could form the basis of an efficient parallel simulation framework.

1 INTRODUCTION

Deployment of large-scale systems is often expensive and sometimes catastrophic as these systems generally have large numbers of interacting components. Failure of these components add uncertainty to the normal operation of the system. To manage this uncertainty, to understand these systems in depth before deployment, and to protect them from unexpected repairs and costs, engineers rely heavily on detailed discrete-event simulations.

Simulation is applied in numerous and diverse fields such as manufacturing systems, communications and protocol design, financial and economic engineering, operations research, design of transportation networks and systems, and so forth. While simulation is a powerful engineering tool for analyzing systems, there are several hurdles before it can be accepted widely as a standard design approach. First, the correctness of the simulation studies depends heavily upon the accuracy of the system representation. Second, simulations of large systems takes a significant computational resources and time. Even though the clock speed of sequential processors improves every year, the complexity of the systems that must be modeled also increases every year. Furthermore, the real utility of simulation lies in comparing alternatives before actual implementation [10]. This suggests that the system model has to be simulated multiple times for a large number of design configurations and parameter values to determine a good design configuration. To perform a thorough analysis of a large number of configurations with varying system design parameter values, it is important to develop efficient simulation methods that can evaluate a large number of system configurations quickly and accurately.

Researchers have focused on evaluating large discrete-event systems using parallel and distributed simulation methods. The novelty and appeal of parallel and distributed simulation methods did not result in the widespread use of these techniques in the real world. This can be attributed solely to the economic viability of the solution for companies, as it was difficult for them to justify their purchase of large clusters of computer nodes to run parallel simulation [12]. To encourage the widespread use of simulation, it is necessary to make simulation more efficient in evaluating large numbers of alternative design choices and configurations. If the system under evaluation is scrutinized carefully, one will notice that changing the system configuration or parameter values does not alter the structure or the behavior dramatically. Rather, much of the system behavior is similar for most of the possible alternative configurations. That fact suggests an efficient way to simulate multiple alternative system configurations simultaneously on a uniprocessor system.

In our previous work on Simultaneous Simulation of Alternative System Configurations (SSASC) [7], we extended the ideas of Vakili [18] and Chen and Ho [4] with a methodology that exploits the structural similarity among the alternative configurations. The result was an efficient simulation algorithm that evaluates all the alternative configurations of a system simultaneously, even when event rates vary greatly, as is often the case in dependability evaluations. While the algorithm is efficient in handling the event list, manipulating the state of the simulated system becomes a bottleneck that negatively impacts speed up. With a large number of simultaneous simulation configurations, updating the state of the entire configuration hits the upper bound of the processors' ability to cache and optimize the execution, thereby reducing the speed up. However, the configurations' states are often very similar, due to the structural similarity between alternative models. Furthermore, state updates follow a unique pattern that allows us to build a data structure that would enable us to represent the states more efficiently, thereby improving the speed up of the SSASC algorithm. The contribution of this paper is the development of an efficient data structure that exploits the unique state update pattern and structural similarity among alternative configurations to enhance the speed up of the simultaneous simulation algorithm. The speedup we obtain is significant, on the average 4-8 times faster than the results presented in [7], and 30-55 times faster than simulation of each of the alternate configurations independently.

The rest of the paper is organized as follows. Section 2 provides an insight into the current state of the art in distributed and parallel simulation. Section 3 describes the data structure we developed to represent the state of simulation model. Analysis and experiment that explore the effectiveness of the algorithm are described in Section 4. We provide an insight to future work in Section 5 and conclude in Section 6.

2 RELATED WORK

Over the last decade, related research in this area has been focused on reducing the cost of the Discrete Event Simulation (DES) and speeding it up using multiprocessors or multiple computers. There are several parallel and distributed simulation (PADS) techniques that use the large number of available processors to increase the simulation efficiency [15]. The speed up gained is due to the number of events handled per unit of time and this gain is limited by the number of processors [11]. Most parallel simulation approaches can be classified into two categories, *conservative* and *optimistic* algorithms, based on how they adhere to local causality constraints. [1] Conservative algorithms never violate the local causality constraint [3]. Optimistic algorithms attempt to exploit the possibility of no causality error [1]. An optimistic algorithm guarantees detection

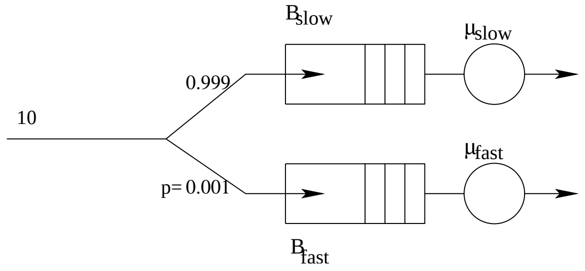

Figure 1: Two-server queuing network.

of such errors and provides a rollback mechanism to restore the simulation to a consistent state. Both techniques improve the efficiency of the simulations, but at the expense of using additional resources.

While efficient simulation of large systems is essential, the real utility of simulation lies in comparing different simulation models that might represent competing or alternative designs before the actual deployment and implementations. The variants of the simulation models are generally systems with different parameter values. Past research by Vakili [18] has looked at simulation of a large number of systems using a technique called the *standard clock*(SC). The SC approach has no event list or event life times. Here, the technique determines the dominant Poisson process among all of the alternative configurations and uses it as the clock rate to clock the simulation. Chen and Ho [4] extended Vakili's work to simulate general distributions by approximating these distributions using shifted exponential and hyper exponentials. More recently, Gaonkar and Sanders [7] developed SSASC by extending Vakili's approach by incorporating adaptive uniformization [16] and a traditional event scheduler while simulating alternative configurations. Heidelberger and Nicol [8] have shown that this approach can be applied in unison with parallel simulation to evaluate systems. Thus, this approach adds onto the existing parallel and distributed simulation framework.

Efficient data structures can also improve the execution time and efficiency of simulation significantly. Biswas and Brown [2], Unger et al. [17], and others have shown how efficient resource management of data structures that hold system state can improve the simulation algorithm. However, their data structures were built to handle the concurrency issues of simulating systems of parallel systems, while we are interested in improving the efficiency of state updates of alternative configurations being simulated simultaneously on a single processor. The previous authors' work showed that unique properties of parallel simulation state update can be utilized to build data structures tailored to the properties that would improve the overall efficiency of simula-

[1]If event *e1* has to be processed before event *e2*, there is a local causality constraint from *e1* to *e2*.

tion [2]. Similarly, we have identified properties that are most helpful in building data structure that can handle state changes/access and updates in SSASC effectively.

3 EFFICIENT STATE MANAGEMENT FOR SSASC

3.1 Background

SSASC[7] has been shown to produce substantial execution speed up due to its efficient event list management. However, there are significant overheads due to other components of the simultaneous simulation implementations. One of the biggest hindrances in achieving the expected speed up is caused by the memory overhead and the execution time of the state saving/updating operation associated with each alternative configuration that is being simulated. Each time an event is fired, the simulation algorithm needs to check and update the state of each alternative configuration. If the number of alternative configurations is large, there is a substantial overhead caused by the need to iterate through the state variable of each simulation configuration and update the state. We define the total state of a simulation model as the cross product of individual states defined in the system. For instance, each place in a petri-net model is a state variable. For ease of discussion, we call individual state as the *state variable* of the model as described in [5].

Table 1: Alternative Design Configurations and State of Simulation Model

Config #	0	1	2	3	4	5	6	7	8	9	10	11
Alternate design configuration parameter values												
B_{slow}	5	5	5	5	5	5	5	5	5	5	5	5
B_{fast}	7	9	7	9	7	9	7	9	7	9	7	9
μ_{slow}	1	1	2	2	1	1	2	2	1	1	2	2
μ_{fast}	3	3	3	3	4	4	4	4	5	5	5	5
Alternate design configuration initial system state												
n_{slow}	0	0	0	0	0	0	0	0	0	0	0	0
n_{fast}	0	0	0	0	0	0	0	0	0	0	0	0

In order to understand the current state management approach in SSASC, consider an example of a two-server queuing system as shown in Figure 1. Customers arrive at a rate of $\lambda = 3$ per second. They are routed to the slow server or the fast server with a probability of 0.001 and 0.999 respectively. Each server has a finite buffer, whose size is denoted by B_{slow} (for the slow server) or B_{fast} (for the fast server). Both servers provide service using the First Come First Serve ($FCFS$) policy. When a job completes, it departs from the system with the maximum state departure rate μ_{slow} or μ_{fast} from the slow or fast server, respectively. The parameter values of B_{slow}, B_{fast}, μ_{slow}, and μ_{fast} are varied to determine alternative competing system designs. The state of each alternative configuration is represented by $< n_{slow}, n_{fast} >$. For simplicity, let us fix the value of B_{slow} to 5 customers, but vary the values of the other parameters, i.e., B_{fast} to 7 and 9 customers, μ_{slow} to 1 and 2 customers per second, and μ_{fast} to 3, 4, and 5 customers per second. We obtain 12 alternative design configurations as shown in Table 1. For the above example, we can represent the state variable of all configurations as a string of 12 integers. Therefore, n_{slow} is 000000000000, and n_{fast} is 000000000000. Table 2 traces out the state variable n_{slow} and n_{fast} for a particular simulation trajectory of the SSASC algorithm.

Table 2: Trace of SSASC simulation of a Two-server queuing network

	Activity (event) fired	State of the system
1	initial state	000000000000 000000000000
2	λ, arrives at slow server	111111111111 000000000000
3	λ, arrives at fast server	111111111111 111111111111
4	μ_{fast}, only configurations with rates 4, 5	111111111111 111100000000
5	λ, arrives at fast server	111111111111 222211111111
6	μ_{slow}, only configurations with rate 2	010101010101 222211111111
7	μ_{fast}	010101010101 111100000000
8	μ_{fast}	010101010101 000000000000

First, for each state variable update, the SSASC has to iterate through each configuration's state and update it individually. That adds a great deal of overhead when the order of the number of configurations is large (often in the thousands). However, in the example, it's evident that the change of state of the different configurations follows a fixed pattern. The reason is that the configurations share similar simulation model structures and very similar stochastic behavioral properties. For example, configuration 0 and 1 differ only in the buffer capacity of the fast server. For the given parameter values, the simulation trajectories of both of them will be almost identical. This gives us the opportunity to design an efficient representation that would reduce cost overhead, both in time and memory, to update the state variable of the configurations.

Second, for each event fired, only a fixed subset of the configurations is updated. From the example, we see that when μ_{fast} fires only for rates 4 and 5 (see line 4 in Table 2), only configurations 4 through 11 were updated. Similarly, for μ_{slow} (see line 6 in Table 2), configurations 1, 3, 5, 7, 9, 11 were updated. All of these patterns can be predetermined before the running of the simulation algorithm to provide a regular structure to the state of the configurations. Doing that could significantly reduce the overhead due to state updating.

Finally, it is important to note that the most common operations on the state variable are accesses and updates to all the alternative configurations. Therefore, the data structure can be designed in a manner that is most efficient for the group access, although it can become more expensive when individual alternative configuration states are accessed or updated.

Using the properties discussed above, we present a data structure and operations supported by it that enables our implementation of SSASC to run significantly faster than our previous implementation.

3.2 Efficient State Management (ESM)

We describe the design of the data structure for efficient state management in two steps. The first step is to construct a data structure that makes it efficient to access and update the state. The second step is to provide additional support for the data structure that enables us to apply the approach described in the first approach to all state variables in a model. For ease of reference, let us call the SSASC algorithm with new state management data-structure as ESM-SSASC.

3.2.1 Individual State

First, we need to represent each state variable in a more compact form. In the SSASC algorithm, each state variable is represented as a linear array of size N, where N is the number of configurations. Each access and update to $n < N$ contiguous configurations requires access to each individual state. Since these configurations are very similar, the state of consecutive states are mostly likely identical during most of the simulation. This allows us to represent the data structure that represents the state as the linked lists in an array. Since the size of that data structure is bounded by the number of alternative configurations, an array implementation of the list is very efficient. Furthermore, updates and accesses are still done on a single contiguous block of memory, which makes it efficient. Each cell in the list has three elements: the *state* of the model, the *index* of the current cell, and the link to the *next* cell. The tuple [*state*, *index*, *next*] represents the state of the system. For example, [0, 2, 5] shows configurations 0 through 5 have 2 customers in the queue.

3.2.2 Ordering of States in the Configurations

Second, in order to maximize the advantage of compact representation as discussed in the previous section, the global order of the configurations should match the sorted order of the event rates on which the state variable depends. If it does not, the array of list that represents the state variables will become fragmented. That, in turn, would make the array of lists large that the obtained speed up would be compromised. From our example, n_{fast} is in the sorted order that matches the order of the configurations. Thus, all states depending only on μ_{fast} will benefit, while state variable n_{slow}, depending on μ_{slow}, will be fragmented, as seen in line 6 of Table 2. This hindrance can be alleviated by building an Indirect Referencing List (IRL) for each state that has a different sorted order of event rates when compared to the actual configuration order. IRL is only built once, during the initialization of the simulation model. IRL is accessed only when the simulation model has operations that have two or more states that interact with each other or when the reward metrics are computed. If models can minimize such restrictions, then significant speed up can be achieved. Therefore, the IRL for n_{fast} is 0,1,2,3,4,5,6,7,8,9,10,11, and the IRL for n_{slow} is 0,2,4,6,8,10,1,3,5,7,9,11. With those additions, the simulation trace seen in Table 2 will be modified as shown in Table 3. With ESM data structure, the updates and accesses are significantly optimized.

Table 3: Trace of SSASC simulation with ESM

	Activity (event) Fired	System State	
1	initial state	[0,0,11]	[0,0,11]
2	λ, arrives at slow server	[1,0,11]	[0,0,11]
3	λ, arrives at fast server	[1,0,11]	[1,0,11]
4	μ_{fast}, only configurations with rates 4, 5	[1,0,11]	[1,0,3],[0,4,11]
5	λ, arrives at fast server	[1,0,11]	[2,0,3],[1,4,11]
6	μ_{slow}, only configurations with rate 2	[0,0,5],[1,6,11]	[2,0,3],[1,4,11]
7	μ_{fast}	[0,0,5],[1,6,11]	[1,0,3],[0,4,11]
8	μ_{fast}	[0,0,5],[1,6,11]	[0,0,11]

4 EXPERIMENTAL EVALUATION

4.1 Analysis

As with any data structure, the efficiency of the execution depends upon the data access/update patterns from the simulation algorithm. In Section 3.1, we looked at some of the state update characteristics and built our data structure to be optimized for those operations. We will now look into some methods that would make the best use of the data structure for the simulation algorithm.

An operation that causes two or more simulation models to directly interact reduces the gained speed up. For instance, customers in slower queue can be transferred to the fast queue once in a while. In the original SSASC algorithm, this would have taken exactly N operations. Here N is the number of alternative configurations and m is the number of interacting state variables. Using ESM would add additional $m * N$ operations due to the usage of IRL.

The order in which the IRL is built for each state variable in the model has some impact on the speedup. Since faster-rate events are fired more often, we achieve

Table 4: Scalability of ESM-SSASC vs Traditional Simulation

Number of alternate configurations	Total Simulation Time (seconds)			Speedup versus Traditional	
	Traditional Simulation	SSASC Simulation	ESM-SSASC Simulation	SSASC Simulation	ESM-SSASC Simulation
1	10.21	10.59	11.16	0.96	0.91
4	41.4	13.17	11.13	3.14	3.72
16	164.9	23.9	12.71	6.90	12.97
64	665.6	68.46	19.17	9.72	34.72
256	2648.0	165.91	47.31	15.96	55.97
1024	10559.0	1617.0	183.10	6.53	57.67
4096	41978.0	8088.0	1688.9	5.19	24.86

better speedup if the list is built based on the order of the dominant event rate that affects each state variable. In situations in which the state variable of a model is affected by more than one event, it is better to build the IRL based on the fastest event affecting the state variable.

In the discussion about state update characteristics in Section 3.1, we noted that minimizing the size of the linear list improves efficiency. The reason is that we could precompute the sorted order of firing of a particular activity for all the alternative configurations. However, note that the state-dependent activity does not guarantee this property of sorted order of firing. Therefore, it might be efficient to avoid using lists and stick to the linear array representation for state variables that depend on state-dependent events.

Finally, reward metrics can be viewed as read-only state variables of the models. Efficiency of the simulation execution can be significantly improved by viewing programming model as N independent alternative simulations that depend on each other for computational efficiency, rather than taking the traditional approach of viewing it as a simulation of N replicated alternative models with different parameter values.

We analyze our simulation technique by studying the availability of an information service system adapted from [14]. The model represents different components of an information service and the interaction and propagation of faults across the components. We evaluate the model for varied configurations and show that our simulation algorithm is efficient and scalable, thus providing a good framework for design optimization techniques. The distributed information service system has a single front-end module that interacts with four processing units. Each processing unit has two processor units, one unit of memory, a switch, and a back-end database. Each of the units can be in any of the following four states: *Working*, *Corrupted*, *Failed*, and *Repaired*. Further details of .the SAN [13] model and the description of the fault model can be found in [7].

4.2 Scalability

Both the simulators, SSASC and ESM-SSASC were built upon the Möbius [5] environment. The efficiency of the simulations were tested on an AMD Athlon XP 2700+ processor running at 2.2 GHz with 1Gb RAM on Fedora Core 3 Operating System. The implementations were compiled using a g++ 3.2.2 with optimization level -03. The experiments were run for 1 million batches for each algorithm. We describe the scalability experiment and the insight we gained from the experiments we conducted in the following paragraph.

Table 4 represents the speed up gained using our new data structure. Profiling the simulation using standard profilers such as gprof [6] reveals that simulator now spends most of the time in computing the model's reward measures. The ESM data structure together with the SSASC algorithm makes the state update and event management efficient for simulating alternate configurations simultaneously.

5 FUTURE WORK

One of the goals of this work is to aid the use of simulation as an objective and/or constraint function in optimization of stochastic systems. Stochastic optimizations are often non-linear and it is not possible to quantify them analytically. That eliminates the possibility of exact calculations of local gradients which traditional optimization solvers rely upon. Our approach provides a way to explore large number of parameter values that could potentially allow us to use alternative approaches, such as compass search, direct search, and other unconstrained optimization more efficiently [9]. Furthermore, as the configurations are simulated, our approach provides a seamless framework that enables us to prune out designs that would not meet the required objectives. The pruning criteria can be defined to prune uninteresting designs to improve the efficiency of the simulations.

SSASC also introduces an incentive to look at a new programming paradigm for describing simulation models. The current programmer's view of a simulation execution sees it as a single model and its behavior. The speed up of this approach is greatly enhanced if the programmer understands programmatic dependencies and interactions between alternative configurations, even though they're stochastically independent. Developing a programming paradigm to maximize the utility of the speed up achieved by the algorithm is also thus interesting research area.

6 CONCLUSIONS

The primary contribution of this paper is a new data structure to manage the state to simulate alternative configurations of dependability models simultaneously We showed in our results that the modified SSASC algorithm improved the speed up significantly relative to the traditional algorithm and also relative to the original SSASC algorithm. We also provided a brief overview and insight into the SSASC algorithm. SSASC algorithm is an approach that can be implemented in conjunction with current implementations of parallel and distributed simulation to allow substantial improvement in simulation efficiency. We expect that our technique will open up new research issues and ideas in simulation optimization, as well as parameter optimization of systems.

ACKNOWLEDGEMENT

This research was supported in part by Hewlett-Packard. We appreciate their support. We would also like to thank Jenny Applequist for her editorial assistance.

REFERENCES

[1] O. Berry and G. Lomow. Speeding up distributed simulation using the time warp mechanism. In *Proceedings of the 2nd Workshop on Making Distributed Systems Work*, pages 1–14, New York, NY, USA, 1986. ACM Press.

[2] J. Biswas and J. C. Browne. Simulation data structures for parallel resource management. *IEEE Trans. Softw. Eng.*, 19(7):672–686, 1993.

[3] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, 1981.

[4] C.-H. Chen and Y.-C. Ho. An approximation approach of the standard-clock method for general discrete-event simulation. *Control Systems Technology*, 3(4):309–318, 1995.

[5] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The m&#246;bius framework and its implementation. *IEEE Trans. Softw. Eng.*, 28(10):956–969, 2002.

[6] S. L. G. et. al. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[7] S. Gaonkar and W. H. Sanders. Simultaneous simulation of alternative system configurations. In *Proceedings of the 11th Pacific Rim Dependable Computing*, pages 41–48, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[8] P. Heidelberger and D. Nicol. Simultaneous parallel simulations of continuous time markov chains at multiple parameter settings. *Proceedings of the 23rd Winter Conference on Winter Simulation*, pages 602–607, 1991.

[9] T. G. Kolda, R. M. Lewis, and V. Torczon. Optimization by direct search: New perspectives on some classical and modern methods. *Society for Industrial and Applied Mathematics (SIAM) Review*, 45(3):385–482, july 2003.

[10] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis.* McGraw Hill, 2000.

[11] Y.-B. Lin. Parallel independent replicated simulation on a network of workstations. In *PADS '94: Proceedings of the Eighth Workshop on Parallel and Distributed Simulation*, pages 73–80, New York, NY, USA, 1994. ACM Press.

[12] Y. Low, C. Lim, W. Cai, S. Huang, W. Hsu, S. Jain, and S. Turner. Survey of languages and runtime libraries for parallel discrete-event simulation. *Simulation*, 72:170–186, March, 1999.

[13] J. F. Meyer, A. Movaghar, and W. H. Sanders. Stochastic activity networks: Structure, behavior, and application. In *Proceedings of the International Workshop on Timed Petri Nets*, pages 106–115, July 1985.

[14] R. R. Muntz and J. Lui. Computing bounds on steady-state availability of repairable computer systems. *Journal of the ACM*, 41(4):676–707, 1994.

[15] D. M. Nicol and R. M. Fujimoto. Parallel simulation today. *Annals of Operations Research*, (53):249–285, 1994.

[16] D. M. Nicol and P. Heidelberger. Parallel simulation of Markovian queueing networks using adaptive uniformization. *SIGMETRICS*, 21(1):135–145, 1993.

[17] B. W. Unger, J. G. Cleary, A. Covington, and D. West. An external state management system for optimistic parallel simulation. In *WSC '93: Proceedings of the 25th Winter Conference on Winter simulation*, pages 750–755, New York, NY, USA, 1993. ACM Press.

[18] P. Vakili. Massively parallel and distributed simulation of a class of discrete event systems: A different perspective. *ACM Trans. Model. Comput. Simul.*, 2(3):214–238, 1992.